



CFD General Notation System Mid-Level Library

Version 2.2.3

Software Release 2.1

Contents

1	Introduction	1
2	General Remarks	3
2.1	Language	3
2.2	Character Strings	3
2.3	Error Status	3
2.4	Typedefs	3
2.5	Acquiring the Software and Documentation	5
2.6	Organization of This Manual	5
3	Opening and Closing a CGNS File	7
4	Navigating a CGNS File	9
5	Error Handling	11
6	Structural Nodes	13
6.1	CGNS Base Information	13
6.2	Zone Information	13
6.3	Simulation Type	14
7	Descriptors	17
7.1	Descriptive Text	17
7.2	Ordinal Value	17
8	Physical Data	19
8.1	Data Arrays	19
8.2	Data Class	20
8.3	Data Conversion Factors	20
8.4	Dimensional Units	21
8.5	Dimensional Exponents	22
9	Location and Position	23
9.1	Grid Location	23
9.2	Rind Layers	23
10	Auxiliary Data	25
10.1	Reference State	25
10.2	Convergence History	25
10.3	Integral Data	26
10.4	User-Defined Data	26
11	Grid Specification	29
11.1	Zone Grid Coordinates	29
11.2	Element Connectivity	31
12	Solution Data	33
12.1	Flow Solution	33
12.2	Discrete Data	35
13	Grid Connectivity	37

13.1 One-to-One Connectivity	37
13.2 Generalized Connectivity	39
13.3 Overset Holes	41
14 Boundary Conditions	43
14.1 Boundary Condition Type and Location	43
14.2 Boundary Condition Datasets	45
14.3 Boundary Condition Data	45
15 Equation Specification	47
15.1 Flow Equation Set	47
15.2 Governing Equations	48
15.3 Auxiliary Models	48
16 Families	51
16.1 Family Definition	51
16.2 Geometry Reference	52
16.3 Family Boundary Condition	53
16.4 Family Name	53
17 Time-Dependent Data	55
17.1 Base Iterative Data	55
17.2 Zone Iterative Data	55
17.3 Rigid Grid Motion	56
17.4 Arbitrary Grid Motion	57
18 Links	59

1 Introduction

This document outlines a CGNS library designed to ease implementation of CGNS by providing developers with a collection of handy I/O functions. Since knowledge of the [ADF core](#) is not required to use this library, it greatly facilitates the task of interfacing with CGNS.

The CGNS Mid-Level Library is based on the [SIDS-to-ADF File Mapping Manual](#). It allows reading and writing all of the information described in that manual including grid coordinates, block interfaces, flow solutions, and boundary conditions. Use of the mid-level library functions insures efficient communication between the user application and the internal representation of the CGNS data.

It is assumed that the reader is familiar with the information in the [CGNS Standard Interface Data Structures \(SIDS\)](#), as well as the [File Mapping Manual](#). The reader is also strongly encouraged to read the [User's Guide to CGNS](#), which contains coding examples using the Mid-Level Library to write and read simple files containing CGNS databases.

2 General Remarks

2.1 Language

The CGNS Mid-Level Library is written in C, but each function has a Fortran counterpart. All function names start with “cg_”. The Fortran functions have the same name as their C counterpart with the addition of the suffix “_f”.

2.2 Character Strings

All data structure names and labels in CGNS are limited to 32 characters. When reading a file, it is advised to pre-allocate the character string variables to 32 characters in Fortran, and 33 in C (to include the string terminator). Other character strings, such as the CGNS file name or descriptor text, are unlimited in length.

2.3 Error Status

All C functions return an integer value representing the error status. All Fortran functions have an additional parameter, `ier`, which contains the value of the error status. An error status different from zero implies that an error occurred. The error message can be printed using the error handling functions of the CGNS library, described in [Section 5](#). The error codes are coded in the C and Fortran include files `cgnslib.h` and `cgnslib.f.h`.

2.4 Typedefs

Several types of variables are defined using typedefs in the `cgnslib.h` file. These are intended to facilitate the implementation of CGNS in C. These variable types are defined as an enumeration of key words admissible for any variable of these types. The file `cgnslib.h` must be included in any C application programs which use these data types.

In Fortran, the same key words are defined as integer parameters in the include file `cgnslib.f.h`. Such variables should be declared as `integer` in Fortran applications. The file `cgnslib.f.h` must be included in any Fortran application using these key words.

The list of supported values (key words) for each of these variable types (typedef) are:

<code>ZoneType_t</code>	Structured, Unstructured
<code>ElementType_t</code>	NODE, BAR_2, BAR_3, TRI_3, TRI_6, QUAD_4, QUAD_8, QUAD_9, TETRA_4, TETRA_10, PYRA_5, PYRA_14, PENTA_6, PENTA_15, PENTA_18, HEXA_8, HEXA_20, HEXA_27, MIXED, NGON_n
<code>DataType_t</code>	Integer, RealSingle, RealDouble, Character
<code>DataClass_t</code>	Dimensional, NormalizedByDimensional, NormalizedByUnknownDimensional, NondimensionalParameter, DimensionlessConstant
<code>MassUnits_t</code>	Null, UserDefined, Kilogram, Gram, Slug, PoundMass

Mid-Level Library

LengthUnits_t Null, UserDefined, Meter, Centimeter, Millimeter, Foot, Inch

TimeUnits_t Null, UserDefined, Second

TemperatureUnits_t
Null, UserDefined, Kelvin, Celsius, Rankine, Fahrenheit

AngleUnits_t Null, UserDefined, Degree, Radian

GoverningEquationsType_t
Null, UserDefined, FullPotential, Euler, NSLaminar, NSTurbulent,
NSLaminarIncompressible, NSTurbulentIncompressible

ModelType_t Null, UserDefined, Ideal, VanderWaals, Constant, PowerLaw,
SutherlandLaw, ConstantPrandtl, EddyViscosity, ReynoldsStress,
ReynoldsStressAlgebraic, Algebraic_BaldwinLomax,
Algebraic_CebeciSmith, HalfEquation_JohnsonKing,
OneEquation_BaldwinBarth, OneEquation_SpalartAllmaras,
TwoEquation_JonesLauder, TwoEquation_MenterSST,
TwoEquation_Wilcox

GridLocation_t Vertex, IFaceCenter, CellCenter, JFaceCenter, FaceCenter,
KFaceCenter, EdgeCenter

GridConnectivityType_t
Overset, Abutting, Abutting1to1

PointSetType_t PointList, PointRange, PointListDonor, PointRangeDonor,
ElementList, ElementRange

BCType_t Null, UserDefined, BCAxisymmetricWedge, BCDegenerateLine,
BCExtrapolate, BCDegeneratePoint, BCDirichlet, BCFarfield,
BCNeumann, BCGeneral, BCInflow, BCOutflow, BCInflowSubsonic,
BCOutflowSubsonic, BCInflowSupersonic, BCOutflowSupersonic,
BCSymmetryPlane, BCTunnelInflow, BCSymmetryPolar,
BCTunnelOutflow, BCWallViscous, BCWall, BCWallViscousHeatFlux,
BCWallInviscid, BCWallViscousIsothermal, FamilySpecified

BCDataType_t Dirichlet, Neumann

RigidGridMotionType_t
Null, UserDefined, ConstantRate, VariableRate

ArbitraryGridMotionType_t
Null, UserDefined, NonDeformingGrid, DeformingGrid

SimulationType_t
TimeAccurate, NonTimeAccurate

Note that these key words need to be written exactly as they appear here, including the use of upper and lower case, to be recognized by the library.

2.5 Acquiring the Software and Documentation

The CGNS Mid-Level Library software may be downloaded from the CGNS web site, located at <http://www.CGNS.org/>. Compiled object code is available for a variety of platforms. Source code and makefiles may also be downloaded.

This manual, as well as the other CGNS documentation, is available in both HTML and PDF format from the CGNS documentation web site, at <http://www.grc.nasa.gov/www/cgns/>.

2.6 Organization of This Manual

The sections that follow describe the Mid-Level Library functions in detail. The first three sections cover opening and closing a CGNS file ([Section 3](#)), accessing a specific node in a CGNS database ([Section 4](#)), and error handling ([Section 5](#)). The remaining sections describe the functions used to read, write, and modify nodes and data in a CGNS database. These sections basically follow the organization used in the “Detailed CGNS Node Descriptions” section of the [SIDS-to-ADF File Mapping Manual](#).

At the start of each sub-section is a *Node* line, listing the applicable CGNS node label.

Next is a table illustrating the syntax for the Mid-Level Library functions. The C functions are shown first, followed by the corresponding Fortran routines. Input variables are shown in an **upright blue** font, and output variables are shown in a *slanted red* font. For each function, the right-hand column lists the modes (read, write, and/or modify) applicable to that function.

The input and output variables are then listed and defined.

3 Opening and Closing a CGNS File

Functions	Modes
<code>ier = cg_open(char *filename, int mode, int *fn);</code>	r w m
<code>ier = cg_version(int fn, float *version);</code>	r w m
<code>ier = cg_close(int fn);</code>	r w m
call <code>cg_open_f(filename, mode, fn, ier)</code>	r w m
call <code>cg_version_f(fn, version, ier)</code>	r w m
call <code>cg_close_f(fn, ier)</code>	r w m

Input/Output

<code>filename</code>	Name of the CGNS file, including path name if necessary. There is no limit on the length of this character variable. (<i>Input</i>)
<code>mode</code>	Mode used for opening the file. The modes currently supported are <code>MODE_READ</code> , <code>MODE_WRITE</code> , and <code>MODE_MODIFY</code> . (<i>Input</i>)
<code>fn</code>	CGNS file index number. (<i>Input</i> for <code>cg_version</code> and <code>cg_close</code> ; <i>output</i> for <code>cg_open</code>)
<code>version</code>	Version number of the CGNS Library. (<i>Output</i>)
<code>ier</code>	Error status. (<i>Output</i>)

The function `cg_open` must always be the first one called. It opens a CGNS file for reading and/or writing and returns an index number `fn`. The index number serves to identify the CGNS file in subsequent function calls. Several CGNS files can be opened simultaneously. The current limit on the number of files opened at once depends on the platform. On an SGI workstation, this limit is set at 100 (parameter `FOPEN_MAX` in `stdio.h`).

The file can be opened in one of the following modes:

<code>MODE_READ</code>	Read only mode.
<code>MODE_WRITE</code>	Write only mode.
<code>MODE_MODIFY</code>	Reading and/or writing is allowed.

The function `cg_close` must always be the last one called. It closes the CGNS file designated by the index number `fn` and frees the memory where the CGNS data was kept. When a file is opened for writing, `cg_close` writes all the CGNS data in memory onto disk prior to closing the file. Consequently, if is omitted, the CGNS file is not written properly.

In order to reduce memory usage and improve execution speed, large arrays such as grid coordinates or flow solutions are not actually stored in memory. Instead, only their ADF ID numbers are kept in memory for future reference. When a CGNS file is open in writing mode, large arrays passed to the library are immediately written into the CGNS file, directly under the root node. When the file is closed, these arrays are moved to their appropriate location in the CGNS tree.

4 Navigating a CGNS File

Functions	Modes
<code>ier = cg_goto(int fn, int B, ..., "end");</code>	r w m
<code>call cg_goto_f(fn, B, ier, ..., 'end')</code>	r w m

Input/Output

- fn** CGNS file index number. (Input)
- B** Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
- ...** Variable argument list used to specify the path to a node. It is composed of an unlimited list of pair-arguments. Each pair-argument takes the form `CGNS_NodeLabel, NodeIndex`. An example is `Zone_t, ZoneIndex`. (Input)
- end** The character string "end" (or 'end' for the Fortran function) must be the last argument. It is used to indicate the end of the argument list. (Input)
- ier** Error status. The possible values, with the corresponding C names (or Fortran parameters) defined in `cgnslib.h` (or `cgnslib.f.h`) are listed below.

<u>Value</u>	<u>Name/Parameter</u>
0	ALL_OK
1	ERROR
2	NODE_NOT_FOUND
3	INCORRECT_PATH

For non-zero values, an error message may be printed using `cg_error_print()`, as described in [Section 5](#). (Output)

This function allows access to any parent-type nodes in a CGNS file. A parent-type node is one that can have children. Nodes that cannot have children, like `Descriptor_t`, are not supported by this function.

Example

```
ier = cg_goto(fn, B, "Zone_t", Z, "FlowSolutions_t", F, "end");
call cg_goto_f(fn, B, ier, 'Zone_t', Z, 'GasModel_t', 1, 'dataArray_t',
              A, 'end')
```


5 Error Handling

Functions	Modes
<code><i>error_message</i> = char *cg_get_error();</code>	r w m
<code>void cg_error_exit();</code>	r w m
<code>void cg_error_print();</code>	r w m
<code>call cg_get_error_f(<i>error_message</i>)</code>	r w m
<code>call cg_error_exit_f()</code>	r w m
<code>call cg_error_print_f()</code>	r w m

If an error occurs during the execution of a CGNS library function, signified by a non-zero value of the error status variable `ier`, an error message may be retrieved using the function `cg_get_error`. The function `cg_error_exit` may then be used to print the error message and stop the execution of the program. Alternatively, `cg_error_print` may be used to print the error message and continue execution of the program.

6 Structural Nodes

6.1 CGNS Base Information

Node: CGNSBase_t

Functions	Modes
<code>ier = cg_base_write(int fn, char *basename, int cell_dim, int phys_dim, int *B);</code>	- w m
<code>ier = cg_nbases(int fn, int *nbases);</code>	r - m
<code>ier = cg_base_read(int fn, int B, char *basename, int *cell_dim, int *phys_dim);</code>	r - m
call <code>cg_base_write_f(fn, basename, cell_dim, phys_dim, B, ier)</code>	- w m
call <code>cg_nbases_f(fn, nbases, ier)</code>	r - m
call <code>cg_base_read_f(fn, B, basename, cell_dim, phys_dim, ier)</code>	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input for <code>cg_base_read</code> ; <i>output</i> for <code>cg_base_write</code>)
nbases	Number of bases present in the CGNS file fn. (<i>Output</i>)
basename	Name of the base. (Input for <code>cg_base_write</code> ; <i>output</i> for <code>cg_base_read</code>)
cell_dim	Dimension of the cells; 3 for volume cells, 2 for surface cells. (Input for <code>cg_base_write</code> ; <i>output</i> for <code>cg_base_read</code>)
phys_dim	Number of coordinates required to define a vector in the field. (Input for <code>cg_base_write</code> ; <i>output</i> for <code>cg_base_read</code>)
ier	Error status. (<i>Output</i>)

6.2 Zone Information

Node: Zone_t

Functions	Modes
<code>ier = cg_zone_write(int fn, int B, char *zonename, int *size, ZoneType_t zonetype, int *Z);</code>	- w m
<code>ier = cg_nzones(int fn, int B, int *nzones);</code>	r - m
<code>ier = cg_zone_read(int fn, int B, int Z, char *zonename, int *size);</code>	r - m
<code>ier = cg_zone_type(int fn, int B, int Z, ZoneType_t *zonetype);</code>	r - m
call <code>cg_zone_write_f(fn, B, zonename, size, zonetype, Z, ier)</code>	- w m
call <code>cg_nzones_f(fn, B, nzones, ier)</code>	r - m
call <code>cg_zone_read_f(fn, B, Z, zonename, size, ier)</code>	r - m
call <code>cg_zone_type_f(fn, B, Z, zonetype, ier)</code>	r - m

Input/Output

- fn** CGNS file index number. (**Input**)
 - B** Base index number, where $1 \leq B \leq \text{nbases}$. (**Input**)
 - Z** Zone index number, where $1 \leq Z \leq \text{nzones}$. (**Input** for `cg_zone_read`,
`cg_zone_type`; *output* for `cg_zone_write`)
 - nzones** Number of zones present in base B. (*Output*)
 - zonename** Name of the zone. (**Input** for `cg_zone_write`; *output* for `cg_zone_read`)
 - size** Number of vertices, cells, and boundary vertices in each (*index*)-dimension. If the nodes are sorted between internal nodes and boundary nodes, then the optional parameter `VertexSizeBoundary` must be set equal to the number of boundary nodes. By default, `VertexSizeBoundary` equals zero meaning that the nodes are unsorted. This option is only useful for unstructured zones. For structured zones, `VertexSizeBoundary` always equals 0 in all directions.
- | <i>Mesh Type</i> | <i>Size</i> |
|------------------|------------------------------------------------------------------------------------------------------------------|
| 3D structured | NVertexI, NVertexJ, NVertexK
NCellI, NCellJ, NCellK
NBoundVertexI = 0, NBoundVertexJ = 0,
NBoundVertexK |
| 2D structured | NVertexI, NVertexJ
NCellI, NCellJ
NBoundVertexI = 0, NBoundVertexJ = 0 |
| Unstructured | NVertex, NCell, NBoundVertex |
- (**Input** for `cg_zone_write`; *output* for `cg_zone_read`)
- zonetype** Type of the zone. The admissible types are `Structured` and `Unstructured`.
(**Input** for `cg_zone_write`; *output* for `cg_zone_type`)
 - ier** Error status. (*Output*)

Note that the zones are sorted alphanumerically to insure that they can always be retrieved in the same order (for the same model).

6.3 Simulation Type

Node: `SimulationType_t`

Functions	Modes
<i>ier</i> = <code>cg_simulation_type_write(int fn, int B, SimulationType_t SimulationType);</code>	- w m
<i>ier</i> = <code>cg_simulation_type_read(int fn, int B, SimulationType_t SimulationType);</code>	r - m
call <code>cg_simulation_type_write_f(fn, B, SimulationType, ier)</code>	- w m
call <code>cg_simulation_type_read_f(fn, B, SimulationType, ier)</code>	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
SimulationType	Type of simulation. Valid types are Null, UserDefined, TimeAccurate, and NonTimeAccurate. (Input for cg_simulation_type_write; <i>output</i> for cg_simulation_type_read)
ier	Error status. (<i>Output</i>)

7 Descriptors

7.1 Descriptive Text

Node: Descriptor_t

Functions	Modes
<i>ier</i> = cg_descriptor_write(char *name, char *text);	- w m
<i>ier</i> = cg_ndescriptors(int *ndescriptors);	r - m
<i>ier</i> = cg_descriptor_read(int D, char *name, char **text);	r - m
call cg_descriptor_write_f(name, text, <i>ier</i>)	- w m
call cg_ndescriptors_f(ndescriptors, <i>ier</i>)	r - m
call cg_descriptor_size_f(D, size, <i>ier</i>)	r - m
call cg_descriptor_read_f(D, name, text, <i>ier</i>)	r - m

Input/Output

ndescriptors	Number of Descriptor_t nodes under the current node. (<i>Output</i>)
D	Descriptor index number, where $1 \leq D \leq \text{ndescriptors}$. (<i>Input</i>)
name	Name of the Descriptor_t node. (<i>Input</i> for cg_descriptor_write; <i>output</i> for cg_descriptor_read)
text	Description held in the Descriptor_t node. (<i>Input</i> for cg_descriptor_write; <i>output</i> for cg_descriptor_read)
size	Size of the descriptor data (Fortran interface only). (<i>Output</i>)
ier	Error status. (<i>Output</i>)

7.2 Ordinal Value

Node: Ordinal_t

Functions	Modes
<i>ier</i> = cg_ordinal_write(int Ordinal);	- w m
<i>ier</i> = cg_ordinal_read(int *Ordinal);	r - m
call cg_ordinal_write_f(Ordinal, <i>ier</i>)	- w m
call cg_ordinal_read_f(Ordinal, <i>ier</i>)	r - m

Input/Output

Ordinal	Integer greater than zero. (<i>Input</i> for cg_ordinal_write; <i>output</i> for cg_ordinal_read)
ier	Error status. (<i>Output</i>)

8 Physical Data

8.1 Data Arrays

Node: DataArray_t

Functions	Modes
<i>ier</i> = cg_array_write(char *ArrayName, DataType_t DataType, int DataDimension, int *DimensionVector, void *Data);	- w m
<i>ier</i> = cg_narrays(int *narrays);	r - m
<i>ier</i> = cg_array_info(int A, char *ArrayName, DataType_t *DataType, int *DataDimension, int *DimensionVector);	r - m
<i>ier</i> = cg_array_read(int A, void *Data);	r - m
<i>ier</i> = cg_array_read_as(int A, DataType_t DataType, void *Data);	r - m
call cg_array_write_f(ArrayName, DataType, DataDimension, DimensionVector, Data, <i>ier</i>)	- w m
call cg_narrays_f(narrays, <i>ier</i>)	r - m
call cg_array_info_f(A, ArrayName, DataType, DataDimension, DimensionVector, <i>ier</i>)	r - m
call cg_array_read_f(A, Data, <i>ier</i>)	r - m
call cg_array_read_as(A, DataType, Data, <i>ier</i>)	r - m

Input/Output

narrays	Number of DataArray_t nodes under the current node. (<i>Output</i>)
A	Data array index, where $1 \leq A \leq \text{narrays}$. (<i>Input</i>)
ArrayName	Name of the DataArray_t node. (<i>Input</i> for cg_array_write; <i>output</i> for cg_array_info)
DataType	Type of data held in the DataArray_t node. The admissible types are Integer, RealSingle, RealDouble, and Character. (<i>Input</i> for cg_array_write, cg_array_read_as; <i>output</i> for cg_array_info)
DataDimension	Number of dimensions. (<i>Input</i> for cg_array_write; <i>output</i> for cg_array_info)
DimensionVector	Number of data elements in each dimension. (<i>Input</i> for cg_array_write; <i>output</i> for cg_array_info)
Data	The data array. (<i>Input</i> for cg_array_write; <i>output</i> for cg_array_read, cg_array_read_as)
ier	Error status. (<i>Output</i>)

8.2 Data Class

Node: DataClass_t

Functions	Modes
<code>ier = cg_dataclass_write(DataClass_t dataclass);</code>	- w m
<code>ier = cg_dataclass_read(DataClass_t *dataclass);</code>	r - m
call <code>cg_dataclass_write_f(dataclass, ier)</code>	- w m
call <code>cg_dataclass_read_f(dataclass, ier)</code>	r - m

Input/Output

`dataclass` Data class for the nodes at this level. See below for the data classes currently supported in CGNS. (*Input* for `cg_dataclass_write`; *output* for `cg_dataclass_read`)

`ier` Error status. (*Output*)

The data classes currently supported in CGNS are:

<code>Dimensional</code>	Regular dimensional data.
<code>NormalizedByDimensional</code>	Nondimensional data that is normalized by dimensional reference quantities.
<code>NormalizedByUnknownDimensional</code>	All fields and reference data are nondimensional.
<code>NondimensionalParameter</code>	Nondimensional parameters such as Mach number and lift coefficient.
<code>DimensionlessConstant</code>	Constant such as π .

These classes are declared within typedef `DataClass_t` in `cgnslib.h`, and as parameters in `cgnslib_f.h`.

8.3 Data Conversion Factors

Node: DataConversion_t

Functions	Modes
<code>ier = cg_conversion_write(DataType_t DataType, void *ConversionScale, void *ConversionOffset);</code>	- w m
<code>ier = cg_conversion_info(DataType_t *DataType);</code>	r - m
<code>ier = cg_conversion_read(void *ConversionScale, void *ConversionOffset);</code>	r - m
call <code>cg_conversion_write_f(DataType, ConversionScale, ConversionOffset, ier)</code>	- w m
call <code>cg_conversion_info_f(DataType, ier)</code>	r - m
call <code>cg_conversion_read_f(ConversionScale, ConversionOffset, ier)</code>	r - m

Input/Output

DataType	Data type in which the conversion factors are recorded. Admissible data types for conversion factors are RealSingle and RealDouble. (Input for cg_conversion_write; output for cg_conversion_info)
ConversionScale	Scaling factor. (Input for cg_conversion_write; output for cg_conversion_read)
ConversionOffset	Offset factor. (Input for cg_conversion_write; output for cg_conversion_read)
ier	Error status. (Output)

The DataConversion_t data structure contains factors to convert the nondimensional data to “raw” dimensional data; these factors are ConversionScale and ConversionOffset. The conversion process is as follows:

$$\text{Data}(\text{raw}) = \text{Data}(\text{nondimensional}) * \text{ConversionScale} + \text{ConversionOffset}$$

8.4 Dimensional Units

Node: DimensionalUnits_t

Functions	Modes
<code>ier = cg_units_write(MassUnits_t mass, LengthUnits_t length, TimeUnits_t time, TemperatureUnits_t temperature, AngleUnits_t angle);</code>	- w m
<code>ier = cg_units_read(MassUnits_t *mass, LengthUnits_t *length, TimeUnits_t *time, TemperatureUnits_t *temperature, AngleUnits_t *angle);</code>	r - m
<code>call cg_units_write_f(mass, length, time, temperature, angle, ier)</code>	- w m
<code>call cg_units_read_f(mass, length, time, temperature, angle, ier)</code>	r - m

Input/Output

mass	Mass units. Admissible values are Null, UserDefined, Kilogram, Gram, Slug, and PoundMass. (Input for cg_units_write; output for cg_units_read)
length	Length units. Admissible values are Null, UserDefined, Meter, Centimeter, Millimeter, Foot, and Inch. (Input for cg_units_write; output for cg_units_read)
time	Time units. Admissible values are Null, UserDefined, and Second. (Input for cg_units_write; output for cg_units_read)
temperature	Temperature units. Admissible values are Null, UserDefined, Kelvin, Celsius, Rankine, and Fahrenheit. (Input for cg_units_write; output for cg_units_read)

Mid-Level Library

- angle** Angle units. Admissible values are Null, UserDefined, Degree, and Radian. (Input for `cg_units_write`; *output* for `cg_units_read`)
- ier** Error status. (*Output*)

The supported units are declared within typedefs in `cgnslib.h` and as parameters in `cgnslib_f.h`.

8.5 Dimensional Exponents

Node: DimensionalExponents_t

Functions	Modes
<i>ier</i> = <code>cg_exponents_write(DataType_t DataType, void *exponents);</code>	- w m
<i>ier</i> = <code>cg_exponents_info(DataType_t *DataType);</code>	r - m
<i>ier</i> = <code>cg_exponents_read(void *exponents);</code>	r - m
call <code>cg_exponents_write_f(DataType, exponents, ier)</code>	- w m
call <code>cg_exponents_info_f(DataType, ier)</code>	r - m
call <code>cg_exponents_read_f(exponents, ier)</code>	r - m

Input/Output

- DataType** Data type in which the exponents are recorded. Admissible data types for the exponents are RealSingle and RealDouble. (Input for `cg_exponents_write`; *output* for `cg_exponents_info`)
- exponents** The five exponents values for the five dimensional units. (Input for `cg_exponents_write`; *output* for `cg_exponents_read`)
- ier** Error status. (*Output*)

9 Location and Position

9.1 Grid Location

Node: GridLocation_t

Functions	Modes
<code>ier = cg_gridlocation_write(GridLocation_t GridLocation);</code>	- w m
<code>ier = cg_gridlocation_read(GridLocation_t *GridLocation);</code>	r - m
call <code>cg_gridlocation_write_f(GridLocation, ier)</code>	- w m
call <code>cg_gridlocation_read_f(GridLocation, ier)</code>	r - m

Input/Output

`GridLocation` Location in the grid. The admissible locations are Null, UserDefined, Vertex, CellCenter, FaceCenter, IFaceCenter, JFaceCenter, KFaceCenter, and EdgeCenter. (**Input** for `cg_gridlocation_write`; *output* for `cg_gridlocation_read`)

`ier` Error status. (*Output*)

9.2 Rind Layers

Node: Rind_t

Functions	Modes
<code>ier = cg_rind_write(int *RindData);</code>	- w m
<code>ier = cg_rind_read(int *RindData);</code>	r - m
call <code>cg_rind_write_f(RindData, ier)</code>	- w m
call <code>cg_rind_read_f(RindData, ier)</code>	r - m

Input/Output

`RindData` Number of rind layers for each computational direction. (**Input** for `cg_rind_write`; *output* for `cg_rind_read`)

`ier` Error status. (*Output*)

10 Auxiliary Data

10.1 Reference State

Node: ReferenceState_t

Functions	Modes
<i>ier</i> = cg_state_write(char *StateDescription);	- w m
<i>ier</i> = cg_state_read(char **StateDescription);	r - m
call cg_state_write_f(StateDescription, <i>ier</i>)	- w m
call cg_state_size_f(Size, <i>ier</i>)	r - m
call cg_state_read_f(StateDescription, <i>ier</i>)	r - m

Input/Output

StateDescription	Text description of reference state. (<i>Input</i> for cg_state_write; <i>output</i> for cg_state_read)
Size	Number of characters in the StateDescription string (Fortran interface only). (<i>Output</i>)
ier	Error status. (<i>Output</i>)

The function `cg_state_write` creates the `ReferenceState_t` node and must be called even if `StateDescription` is undefined (i.e., a blank string). The descriptors, data arrays, data class, and dimensional units characterizing the `ReferenceState_t` data structure may be added to this data structure after its creation.

The function `cg_state_read` reads the `StateDescription` of the local `ReferenceState_t` node. If `StateDescription` is undefined in the CGNS database, this function returns a null string.

10.2 Convergence History

Node: ConvergenceHistory_t

Functions	Modes
<i>ier</i> = cg_convergence_write(int niterations, char *NormDefinitions);	- w m
<i>ier</i> = cg_convergence_read(int *niterations, char **NormDefinitions);	r - m
call cg_convergence_write_f(niterations, NormDefinitions, <i>ier</i>)	- w m
call cg_convergence_read_f(niterations, NormDefinitions, <i>ier</i>)	r - m

Input/Output

niterations	Number of iterations for which convergence information is recorded. (<i>Input</i> for cg_convergence_write; <i>output</i> for cg_convergence_read)
NormDefinitions	Description of the convergence information recorded in the data arrays. (<i>Input</i> for cg_convergence_write; <i>output</i> for cg_convergence_read)

Mid-Level Library

`ier` Error status. (*Output*)

The function `cg_convergence_write` creates a `ConvergenceHistory_t` node. It must be the first one called when recording convergence history data. The `NormDefinitions` may be left undefined (i.e., a blank string). After creation of this node, the descriptors, data arrays, data class, and dimensional units characterizing the `ConvergenceHistory_t` data structure may be added.

The function `cg_convergence_read` reads a `ConvergenceHistory_t` node. If `NormDefinitions` is not defined in the CGNS database, this function returns a null string.

10.3 Integral Data

Node: `IntegralData_t`

Functions	Modes
<code>ier = cg_integral_write(char *Name);</code>	- w m
<code>ier = cg_nintegrals(int *nintegrals);</code>	r - m
<code>ier = cg_integral_read(int Index, char *Name);</code>	r - m
call <code>cg_integral_write_f(Name, ier)</code>	- w m
call <code>cg_nintegrals_f(nintegrals, ier)</code>	r - m
call <code>cg_integral_read_f(Index, Name, ier)</code>	r - m

Input/Output

`Name` Name of the `IntegralData_t` data structure. (*Input* for `cg_integral_write`; *output* for `cg_integral_read`)

`nintegrals` Number of `IntegralData_t` nodes under current node. (*Output*)

`Index` Integral data index number, where $1 \leq \text{Index} \leq \text{nintegrals}$. (*Input*)

`ier` Error status. (*Output*)

10.4 User-Defined Data

Node: `UserDefinedData_t`

Functions	Modes
<code>ier = cg_user_data_write(char *Name);</code>	- w m
<code>ier = cg_nuser_data(int *nuserdata);</code>	r - m
<code>ier = cg_user_data_read(int Index, char *Name);</code>	r - m
call <code>cg_user_data_write_f(Name, ier)</code>	- w m
call <code>cg_nuser_data_f(nuserdata, ier)</code>	r - m
call <code>cg_user_data_read_f(Index, Name, ier)</code>	r - m

Input/Output

<code>nuserdata</code>	Number of <code>UserDefinedData_t</code> nodes under current node. (<i>Output</i>)
<code>Name</code>	Name of the <code>UserDefinedData_t</code> node. (<i>Input</i> for <code>cg_user_data_write</code> ; <i>output</i> for <code>cg_user_data_read</code>)
<code>Index</code>	User-defined data index number, where $1 \leq \text{Index} \leq \text{nuserdata}$. (<i>Input</i>)
<code>ier</code>	Error status. (<i>Output</i>)

11 Grid Specification

11.1 Zone Grid Coordinates

Node: GridCoordinates_t

GridCoordinates_t nodes are used to describe grids associated with a particular zone. The original grid must be described by a GridCoordinates_t node named GridCoordinates. Additional GridCoordinates_t nodes may be used, with user-defined names, to store grids at multiple time steps or iterations. In addition to the discussion of the GridCoordinates_t node in the [SIDS](#) and [File Mapping](#) manuals, see the discussion of the ZoneIterativeData_t and ArbitraryGridMotion_t nodes in the SIDS manual.

The three functions described below are applicable to any GridCoordinates_t node.

Functions	Modes
<code>ier = cg_grid_write(int fn, int B, int Z, char *GridCoordName, int *G);</code>	- w m
<code>ier = cg_ngrids(int fn, int B, int Z, int *ngrids);</code>	- w m
<code>ier = cg_grid_read(int fn, int B, int Z, int G, char *GridCoordName);</code>	r - m
call <code>cg_grid_write_f(fn, B, Z, GridCoordName, G, ier)</code>	- w m
call <code>cg_ngrids_f(fn, B, Z, ngrids, ier)</code>	- w m
call <code>cg_grid_read_f(fn, B, Z, G, GridCoordName, ier)</code>	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
G	Grid index number, where $1 \leq G \leq \text{ngrids}$. (Input for <code>cg_grid_read</code> ; output for <code>cg_grid_write</code>)
ngrids	Number of GridCoordinates_t nodes for zone Z. (Output)
GridCoordinateName	Name of the GridCoordinates_t node. Note that the name "GridCoordinates" is reserved for the original grid and must be the first GridCoordinates_t node to be defined. (Input for <code>cg_grid_write</code> ; output for <code>cg_grid_read</code>)
ier	Error status. (Output)

The following functions are applicable *only* to the GridCoordinates_t node named GridCoordinates, used for the original grid in a zone. Coordinates for additional GridCoordinates_t nodes in a zone must be read and written using the `cg_array_xxx` functions described in [Section 8.1](#).

Mid-Level Library

Functions	Modes
<code>ier = cg_coord_write(int fn, int B, int Z, DataType_t datatype, char *coordname, void *coord_array, int *C);</code>	- w m
<code>ier = cg_ncoords(int fn, int B, int Z, int *ncoords);</code>	r - m
<code>ier = cg_coord_info(int fn, int B, int Z, int C, DataType_t *datatype, char *coordname);</code>	r - m
<code>ier = cg_coord_read(int fn, int B, int Z, char *coordname, DataType_t datatype, int *range_min, int *range_max, void *coord_array);</code>	r - m
call <code>cg_coord_write_f(fn, B, Z, datatype, coordname, coord_array, C, ier)</code>	- w m
call <code>cg_ncoords_f(fn, B, Z, ncoords, ier)</code>	r - m
call <code>cg_coord_info_f(fn, B, Z, C, datatype, coordname, ier)</code>	r - m
call <code>cg_coord_read_f(fn, B, Z, coordname, datatype, range_min, range_max, coord_array, ier)</code>	r - m

Input/Output

<code>fn</code>	CGNS file index number. (Input)
<code>B</code>	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
<code>Z</code>	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
<code>C</code>	Coordinate array index number, where $1 \leq C \leq \text{ncoords}$. (Input for <code>cg_coord_info</code> ; <i>output</i> for <code>cg_coord_write</code>)
<code>ncoords</code>	Number of coordinate arrays for zone Z. (<i>Output</i>)
<code>datatype</code>	Data type in which the coordinate array is written. Admissible data types for a coordinate array are <code>RealSingle</code> and <code>RealDouble</code> . (Input for <code>cg_coord_write</code> , <code>cg_coord_read</code> ; <i>output</i> for <code>cg_coord_info</code>)
<code>coordname</code>	Name of the coordinate array. It is strongly advised to use the SIDS nomenclature conventions when naming the coordinate arrays to insure file compatibility. (Input for <code>cg_coord_write</code> , <code>cg_coord_read</code> ; <i>output</i> for <code>cg_coord_info</code>)
<code>range_min</code>	Lower range index (eg., <code>imin</code> , <code>jmin</code> , <code>kmin</code>). (Input)
<code>range_max</code>	Upper range index (eg., <code>imax</code> , <code>jmax</code> , <code>kmax</code>). (Input)
<code>coord_array</code>	Array of coordinate values for the range prescribed. (Input for <code>cg_coord_write</code> ; <i>output</i> for <code>cg_coord_read</code>)
<code>ier</code>	Error status. (<i>Output</i>)

11.2 Element Connectivity

Node: Elements_t

Functions	Modes
<code>ier = cg_section_write(int fn, int B, int Z, char *ElementSectionName, ElementType_t type, int start, int end, int nbndry, int *Elements, int *S);</code>	- w m
<code>ier = cg_parent_data_write(int fn, int B, int Z, int S, int *ParentData);</code>	- w m
<code>ier = cg_nsections(int fn, int B, int Z, int *nsections);</code>	r - m
<code>ier = cg_section_read(int fn, int B, int Z, int S, char *ElementSectionName, ElementType_t *type, int *start, int *end, int *nbndry, int *parent_flag);</code>	r - m
<code>ier = cg_ElementDataSize(int fn, int B, int Z, int S, int *ElementDataSize);</code>	r - m
<code>ier = cg_elements_read(int fn, int B, int Z, int S, int *Elements, int *ParentData);</code>	r - m
<code>ier = cg_npe(ElementType_t type, int *npe);</code>	r w m
call <code>cg_section_write_f(fn, B, Z, ElementSectionName, type, start, end, nbndry, Elements, S, ier)</code>	- w m
call <code>cg_parent_data_write_f(fn, B, Z, S, ParentData, ier)</code>	- w m
call <code>cg_nsections_f(fn, B, Z, nsections, ier)</code>	r - m
call <code>cg_section_read_f(fn, B, Z, S, ElementSectionName, type, start, end, nbndry, parent_flag, ier)</code>	r - m
call <code>cg_ElementDataSize_f(fn, B, Z, S, ElementDataSize, ier)</code>	r - m
call <code>cg_elements_read_f(fn, B, Z, S, Elements, ParentData, ier)</code>	r - m
call <code>cg_npe_f(type, npe, ier)</code>	r w m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
ElementSectionName	Name of the Elements_t node. (Input for <code>cg_section_write</code> ; <i>output</i> for <code>cg_section_read</code>)
type	Type of element. See the eligible types for ElementType_t in Section 2.4. (Input for <code>cg_section_write</code> , <code>cg_npe</code> ; <i>output</i> for <code>cg_section_read</code>)
start	Index of first element in the section. (Input for <code>cg_section_write</code> ; <i>output</i> for <code>cg_section_read</code>)
end	Index of last element in the section. (Input for <code>cg_section_write</code> ; <i>output</i> for <code>cg_section_read</code>)

Mid-Level Library

<code>nbndry</code>	Index of last boundary element in the section. Set to zero if the elements are unsorted. (Input for <code>cg_section_write</code> ; <i>output</i> for <code>cg_section_read</code>)
<code>nsections</code>	Lower range index (eg., <code>imin</code> , <code>jmin</code> , <code>kmin</code>). (<i>Output</i>)
<code>S</code>	Element section index, where $1 \leq S \leq nsections$. (Input for <code>cg_parent_data_write</code> , <code>cg_section_read</code> , <code>cg_ElementDataSize</code> , <code>cg_elements_read</code> ; <i>output</i> for <code>cg_section_write</code>)
<code>parent_flag</code>	Flag indicating if the parent data are defined. If the parent data exist, <code>parent_flag</code> is set to 1; otherwise it is set to 0. (<i>Output</i>)
<code>ElementDataSize</code>	Number of element connectivity data values. (<i>Output</i>)
<code>Elements</code>	Element connectivity data. (Input for <code>cg_section_write</code> ; <i>output</i> for <code>cg_elements_read</code>)
<code>ParentData</code>	For boundary or interface elements, this array contains information on the cell(s) and cell face(s) sharing the element. (<i>Output</i>)
<code>npe</code>	Number of nodes for an element of type <code>type</code> . (<i>Output</i>)
<code>ier</code>	Error status. (<i>Output</i>)

12 Solution Data

12.1 Flow Solution

Node: FlowSolution_t

The three functions described below are used to create, and get information about, FlowSolution_t nodes.

Functions	Modes
<i>ier</i> = cg_sol_write(int <i>fn</i> , int <i>B</i> , int <i>Z</i> , char * <i>solname</i> , GridLocationType_t <i>location</i> , int * <i>S</i>);	- w m
<i>ier</i> = cg_nsols(int <i>fn</i> , int <i>B</i> , int <i>Z</i> , int * <i>nsols</i>);	r - m
<i>ier</i> = cg_sol_info(int <i>fn</i> , int <i>B</i> , int <i>Z</i> , int <i>S</i> , char * <i>solname</i> , GridLocationType_t * <i>location</i>);	r - m
call cg_sol_write_f(<i>fn</i> , <i>B</i> , <i>Z</i> , <i>solname</i> , <i>location</i> , <i>S</i> , <i>ier</i>)	- w m
call cg_nsols_f(<i>fn</i> , <i>B</i> , <i>Z</i> , <i>nsols</i> , <i>ier</i>)	r - m
call cg_sol_info_f(<i>fn</i> , <i>B</i> , <i>Z</i> , <i>S</i> , <i>solname</i> , <i>location</i> , <i>ier</i>)	r - m

Input/Output

<i>fn</i>	CGNS file index number. (Input)
<i>B</i>	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
<i>Z</i>	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
<i>S</i>	Flow solution index number, where $1 \leq S \leq \text{nsols}$. (Input for cg_sol_info; <i>output</i> for cg_sol_write)
<i>nsols</i>	Number of flow solutions for zone <i>Z</i> . (<i>Output</i>)
<i>solname</i>	Name of the flow solution. (Input for cg_sol_write; <i>output</i> for cg_sol_info)
<i>location</i>	Grid location where the solution is recorded. The current admissible locations are Vertex, CellCenter, IFaceCenter, JFaceCenter, and KFaceCenter. (Input for cg_sol_write; <i>output</i> for cg_sol_info)
<i>ier</i>	Error status. (<i>Output</i>)

The following functions are used to read and write solution arrays stored below a FlowSolution_t node.

Mid-Level Library

Functions	Modes
<i>ier</i> = cg_field_write(int fn, int B, int Z, int S, DataType_t datatype, char *fieldname, void *solution_array, int *F);	- w m
<i>ier</i> = cg_nfields(int fn, int B, int Z, int S, int *nfields);	r - m
<i>ier</i> = cg_field_info(int fn, int B, int Z, int S, int F, DataType_t *datatype, char *fieldname);	r - m
<i>ier</i> = cg_field_read(int fn, int B, int Z, int S, char *fieldname, DataType_t datatype, int *range_min, int *range_max, void *solution_array);	r - m
call cg_field_write_f(fn, B, Z, S, datatype, fieldname, solution_array, F, <i>ier</i>)	- w m
call cg_nfields_f(fn, B, Z, S, <i>nfields</i> , <i>ier</i>)	r - m
call cg_field_info_f(fn, B, Z, S, F, <i>datatype</i> , <i>fieldname</i> , <i>ier</i>)	r - m
call cg_field_read_f(fn, B, Z, S, <i>fieldname</i> , <i>datatype</i> , <i>range_min</i> , <i>range_max</i> , <i>solution_array</i> , <i>ier</i>)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
S	Flow solution index number, where $1 \leq S \leq \text{nsols}$. (Input)
F	Solution array index number, where $1 \leq F \leq \text{nfields}$. (Input for cg_field_info; output for cg_field_write)
nfields	Number of data arrays in flow solution S. (Output)
datatype	Data type in which the solution array is written. Admissible data types for a solution array are Integer, RealSingle, and RealDouble. (Input for cg_field_write, cg_field_read; output for cg_field_info)
fieldname	Name of the solution array. It is strongly advised to use the SIDS nomenclature conventions when naming the solution arrays to insure file compatibility. (Input for cg_field_write, cg_field_read; output for cg_field_info)
range_min	Lower range index (eg., imin, jmin, kmin). (Input)
range_max	Upper range index (eg., imax, jmax, kmax). (Input)
solution_array	Array of solution values for the range prescribed. (Input for cg_field_write; output for cg_field_read)
ier	Error status. (Output)

The function cg_field_read returns the solution array fieldname, for the range prescribed by range_min and range_max. The array is returned to the application in the data type requested in datatype. This data type does not need to be the same as the one in which the data is stored in the

file. A solution array stored as double precision in the CGNS file can be returned to the application as single precision, or vice versa.

In Fortran, if the variable `solution_array` is defined as a 2D or 3D array, the user must ensure that the dimension allocated for each direction matches exactly the dimension required by the range requested. For example, to read all solution values of a 2D zone with 3×5 vertices, the array must be declared with the size (3,5). If the array is declared bigger (eg., (6,9)) the returned array values will be wrong. To avoid such problems, the array can simply be allocated as a 1D array (eg., (15)). When using a 1D array, the declared size can be larger than the number of data requested.

12.2 Discrete Data

Node: DiscreteData_t

DiscreteData_t nodes are intended for the storage of fields of data not usually identified as part of the flow solution, such as fluxes or equation residuals.

Functions	Modes
<code>ier = cg_discrete_write(int fn, int B, int Z, char *DiscreteName, int *D);</code>	- w m
<code>ier = cg_ndiscrete(int fn, int B, int Z, int *ndiscrete);</code>	r - m
<code>ier = cg_discrete_read(int fn, int B, int Z, int D, char *DiscreteName);</code>	r - m
call <code>cg_discrete_write_f(fn, B, Z, DiscreteName, D, ier)</code>	- w m
call <code>cg_ndiscrete_f(fn, B, Z, ndiscrete, ier)</code>	r - m
call <code>cg_discrete_read_f(fn, B, Z, D, DiscreteName, ier)</code>	r - m

Input/Output

<code>fn</code>	CGNS file index number. (<i>Input</i>)
<code>B</code>	Base index number, where $1 \leq B \leq \text{nbases}$. (<i>Input</i>)
<code>Z</code>	Zone index number, where $1 \leq Z \leq \text{nzones}$. (<i>Input</i>)
<code>D</code>	Discrete data index number, where $1 \leq D \leq \text{ndiscrete}$. (<i>Input</i> for <code>cg_discrete_read</code> ; <i>output</i> for <code>cg_discrete_write</code>)
<code>ndiscrete</code>	Number of DiscreteData_t data structures under zone Z. (<i>Output</i>)
<code>DiscreteName</code>	Name of DiscreteData_t data structure. (<i>Input</i> for <code>cg_discrete_write</code> ; <i>output</i> for <code>cg_discrete_read</code>)
<code>ier</code>	Error status. (<i>Output</i>)

13 Grid Connectivity

13.1 One-to-One Connectivity

Node: GridConnectivity1to1_t

The two functions described below may be used to get information about all the one-to-one zone interfaces in a CGNS database.

Functions	Modes
<code>ier = cg_n1to1_global(int fn, int B, int *n1to1_global);</code>	r - m
<code>ier = cg_1to1_read_global(int fn, int B, char **connectname, char **zonename, char **donorname, int **range, int **donor_range, int **transform);</code>	r - m
call <code>cg_n1to1_global_f(fn, B, n1to1_global, ier)</code>	r - m
call <code>cg_1to1_read_global_f(fn, B, connectname, zonename, donorname, range, donor_range, transform, ier)</code>	r - m

Input/Output

<code>fn</code>	CGNS file index number. (<i>Input</i>)
<code>B</code>	Base index number, where $1 \leq B \leq \text{nbases}$. (<i>Input</i>)
<code>n1to1_global</code>	Total number of one-to-one interfaces in base B, stored under <code>GridConnectivity1to1_t</code> nodes. (I.e., this does not include one-to-one interfaces that may be stored under <code>GridConnectivity_t</code> nodes, used for generalized zone interfaces.) Note that the function <code>cg_n1to1</code> (described below) may be used to get the number of one-to-one interfaces in a specific zone. (<i>Output</i>)
<code>connectname</code>	Name of the interface. (<i>Output</i>)
<code>zonename</code>	Name of the first zone, for all one-to-one interfaces in base B. (<i>Output</i>)
<code>donorname</code>	Name of the second zone, for all one-to-one interfaces in base B. (<i>Output</i>)
<code>range</code>	Range of points for the first zone, for all one-to-one interfaces in base B. (<i>Output</i>)
<code>donor_range</code>	Range of points for the current zone, for all one-to-one interfaces in base B. (<i>Output</i>)
<code>transform</code>	Short hand notation for the transformation matrix defining the relative orientation of the two zones. This transformation is given for all one-to-one interfaces in base B. See the description of <code>GridConnectivity1to1_t</code> in the SIDS manual for details. (<i>Output</i>)
<code>ier</code>	Error status. (<i>Output</i>)

The following functions are used to read and write one-to-one connectivity data for a specific zone.

Mid-Level Library

Functions	Modes
<pre>ier = cg_1to1_write(int fn, int B, int Z, char *connectname, char *donorname, int *range, int *donor_range, int *transform, int *I);</pre>	- w m
<pre>ier = cg_n1to1(int fn, int B, int Z, int *n1to1);</pre>	r - m
<pre>ier = cg_1to1_read(int fn, int B, int Z, int I, char *connectname, char *donorname, int *range, int *donor_range, int *transform);</pre>	r - m
<pre>call cg_1to1_write_f(fn, B, Z, connectname, donorname, range, donor_range, transform, I, ier)</pre>	- w m
<pre>call cg_n1to1_f(fn, B, Z, n1to1, ier)</pre>	r - m
<pre>call cg_1to1_read_f(fn, B, Z, I, connectname, donorname, range, donor_range, transform, ier)</pre>	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
I	Interface index number, where $1 \leq I \leq \text{n1to1}$. (Input for <code>cg_1to1_read</code> ; <i>output</i> for <code>cg_1to1_write</code>)
n1to1	Number of one-to-one interfaces in zone Z, stored under <code>GridConnectivity1to1_t</code> nodes. (I.e., this does not include one-to-one interfaces that may be stored under <code>GridConnectivity_t</code> nodes, used for generalized zone interfaces.) (<i>Output</i>)
connectname	Name of the interface. (Input for <code>cg_1to1_write</code> ; <i>output</i> for <code>cg_1to1_read</code>)
donorname	Name of the zone interfacing with the current zone. (Input for <code>cg_1to1_write</code> ; <i>output</i> for <code>cg_1to1_read</code>)
range	Range of points for the current zone. (Input for <code>cg_1to1_write</code> ; <i>output</i> for <code>cg_1to1_read</code>)
donor_range	Range of points for the donor zone. (Input for <code>cg_1to1_write</code> ; <i>output</i> for <code>cg_1to1_read</code>)
transform	Short hand notation for the transformation matrix defining the relative orientation of the two zones. See the description of <code>GridConnectivity1to1_t</code> in the SIDS manual for details. (Input for <code>cg_1to1_write</code> ; <i>output</i> for <code>cg_1to1_read</code>)
ier	Error status. (<i>Output</i>)

13.2 Generalized Connectivity

Node: GridConnectivity_t

Functions	Modes
<pre>ier = cg_conn_write(int fn, int B, int Z, char *connectname, GridLocationType_t location, GridConnectivityType_t connect_type, PointSetType_t ptset_type, int npnts, int *pnts, char *donorname, ZoneType_t donor_zonetype, PointSetType_t donor_ptset_type, DataType_t donor_datatype, int ndata_donor, void *donor_data, int *I);</pre>	- w m
<pre>ier = cg_nconns(int fn, int B, int Z, int *nconns);</pre>	r - m
<pre>ier = cg_conn_info(int fn, int B, int Z, int I, char *connectname, GridLocationType_t *location, GridConnectivityType_t *connect_type, PointSetType_t *ptset_type, int *npnts, char *donorname, ZoneType_t *donor_zonetype, PointSetType_t *donor_ptset_type, DataType_t *donor_datatype, int *ndata_donor);</pre>	r - m
<pre>ier = cg_conn_read(int fn, int B, int Z, int I, int *pnts, DataType_t donor_datatype, void *donor_data);</pre>	r - m
<pre>call cg_conn_write_f(fn, B, Z, connectname, location, connect_type, ptset_type, npnts, pnts, donorname, donor_zonetype, donor_ptset_type, donor_datatype, ndata_donor, donor_data, I, ier)</pre>	- w m
<pre>call cg_nconns_f(fn, B, Z, nconns, ier)</pre>	r - m
<pre>call cg_conn_info_f(fn, B, Z, I, connectname, location, connect_type, ptset_type, npnts, donorname, donor_zonetype, donor_ptset_type, donor_datatype, ndata_donor, ier)</pre>	r - m
<pre>call cg_conn_read_f(fn, B, Z, I, pnts, donor_datatype, donor_data, ier)</pre>	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
I	Discrete data index number, where $1 \leq I \leq \text{nconns}$. (Input for <code>cg_conn_info</code> , <code>cg_conn_read</code> ; <i>output</i> for <code>cg_conn_write</code>)
nconns	Number of interfaces for zone Z. (<i>Output</i>)
connectname	Name of the interface. (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)

Mid-Level Library

<code>location</code>	Grid location used in the definition of the point set. The currently admissible locations are <code>Vertex</code> and <code>CellCenter</code> . (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>connect_type</code>	Type of interface being defined. The admissible types are <code>Overset</code> , <code>Abutting</code> , and <code>Abutting1to1</code> . (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>ptset_type</code>	Type of point set defining the interface in the current zone; either <code>PointRange</code> or <code>PointList</code> . (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>donor_ptset_type</code>	Type of point set defining the interface in the donor zone; either <code>PointListDonor</code> or <code>CellListDonor</code> . (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>npnts</code>	Number of points defining the interface in the current zone. For a <code>ptset_type</code> of <code>PointRange</code> , <code>npnts</code> is always two. For a <code>ptset_type</code> of <code>PointList</code> , <code>npnts</code> is the number of points in the <code>PointList</code> . (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>ndata_donor</code>	Number of points or cells defining the interface in the donor zone. (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>donorname</code>	Name of the zone interfacing with the current zone. (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>donor_datatype</code>	Data type in which the donor points are stored in the file. The only admissible type, as of version 2.0 of the Mid-Level Library, is <code>Integer</code> . The <code>donor_datatype</code> argument was left in these functions only for backward compatibility. (Input for <code>cg_conn_write</code> , <code>cg_conn_read</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>pnts</code>	Array of points defining the interface in the current zone. (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_read</code>)
<code>donor_data</code>	Array of points or cells defining the interface in the donor zone. (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_read</code>)
<code>donor_zonetype</code>	Type of the donor zone. The admissible types are <code>Structured</code> and <code>Unstructured</code> . (Input for <code>cg_conn_write</code> ; <i>output</i> for <code>cg_conn_info</code>)
<code>ier</code>	Error status. (<i>Output</i>)

Note that the interpolation factors stored in the `InterpolantsDonor` data array are accessed using the `cg_goto` and `cg_array_xxx` functions, described in [Section 4](#) and [Section 8.1](#), respectively.

13.3 Overset Holes

Node: OversetHoles_t

Functions	Modes
<i>ier</i> = cg_hole_write(int fn, int B, int Z, char *holename, GridLocationType_t location, PointSetType_t ptset_type, int nptsets, int npnts, int *pnts, int *I);	- w m
<i>ier</i> = cg_nholes(int fn, int B, int Z, int *nholes);	r - m
<i>ier</i> = cg_hole_info(int fn, int B, int Z, int I, char *holename, GridLocationType_t *location, PointSetType_t *ptset_type, int *nptsets, int *npnts);	r - m
<i>ier</i> = cg_hole_read(int fn, int B, int Z, int I, int *pnts);	r - m
call cg_hole_write_f(fn, B, Z, holename, location, ptset_type, nptsets, npnts, pnts, I, ier)	- w m
call cg_nholes_f(fn, B, Z, nholes, ier)	r - m
call cg_hole_info_f(fn, B, Z, I, holename, location, ptset_type, nptsets, npnts, ier)	r - m
call cg_hole_read_f(fn, B, Z, I, pnts, ier)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
I	Overset hole index number, where $1 \leq I \leq \text{nholes}$. (Input for cg_hole_info, cg_hole_read; output for cg_hole_write)
nholes	Number of overset holes in zone Z. (Output)
holename	Name of the overset hole. (Input for cg_hole_write; output for cg_hole_info)
location	Grid location used in the definition of the point set. The currently admissible locations are Vertex and CellCenter. (Input for cg_hole_write; output for cg_hole_info)
ptset_type	The extent of the overset hole may be defined using a range of points or cells, or using a discrete list of all points or cells in the overset hole. If a range of points or cells is used, ptset_type is set to PointRange. When a discrete list of points or cells is used, ptset_type equals PointList. (Input for cg_hole_write; output for cg_hole_info)
nptsets	Number of point sets used to define the hole. If ptset_type is PointRange, several point sets may be used. If ptset_type is PointList, only one point set is allowed. (Input for cg_hole_write; output for cg_hole_info)

Mid-Level Library

<code>npnts</code>	Number of points (or cells) in the point set. For a <code>ptset_type</code> of <code>PointRange</code> , <code>npnts</code> is always two. For a <code>ptset_type</code> of <code>PointList</code> , <code>npnts</code> is the number of points or cells in the <code>PointList</code> . (<i>Input</i> for <code>cg_hole_write</code> ; <i>output</i> for <code>cg_hole_info</code>)
<code>pnts</code>	Array of points or cells in the point set. (<i>Input</i> for <code>cg_hole_write</code> ; <i>output</i> for <code>cg_hole_read</code>)
<code>ier</code>	Error status. (<i>Output</i>)

14 Boundary Conditions

14.1 Boundary Condition Type and Location

Node: BC_t

Functions	Modes
<code>ier = cg_boco_write(int fn, int B, int Z, char *boconame, BType_t bocotype, PointSetType_t ptset_type, int npnts, int *pnts, int *BC);</code>	- w m
<code>ier = cg_boco_normal_write(int fn, int B, int Z, int BC, int *NormalIndex, int NormalListFlag, DataType_t NormalDataType, void *NormalList);</code>	- w m
<code>ier = cg_nbocos(int fn, int B, int Z, int *nbocos);</code>	r - m
<code>ier = cg_boco_info(int fn, int B, int Z, int BC, char *boconame, BType_t *bocotype, PointSetType_t *ptset_type, int *npnts, int *NormalIndex, int *NormalListFlag, DataType_t *NormalDataType, int *ndataset);</code>	r - m
<code>ier = cg_boco_read(int fn, int B, int Z, int BC, int *pnts, void *NormalList);</code>	r - m
call <code>cg_boco_write_f(fn, B, Z, boconame, bocotype, ptset_type, npnts, pnts, BC, ier)</code>	- w m
call <code>cg_boco_normal_write_f(fn, B, Z, BC, NormalIndex, NormalListFlag, NormalDataType, NormalList, ier)</code>	- w m
call <code>cg_nbocos_f(fn, B, Z, nbocos, ier)</code>	r - m
call <code>cg_boco_info_f(fn, B, Z, BC, boconame, bocotype, ptset_type, npnts, NormalIndex, NormalListFlag, NormalDataType, ndataset, ier)</code>	r - m
call <code>cg_boco_read_f(fn, B, Z, BC, pnts, NormalList, ier)</code>	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
BC	Boundary condition index number, where $1 \leq BC \leq \text{nbocos}$. (Input for <code>cg_boco_normal_write</code> , <code>cg_boco_info</code> , <code>cg_boco_read</code> ; <i>output</i> for <code>cg_boco_write</code>)
nbocos	Number of boundary conditions in zone Z. (<i>Output</i>)
boconame	Name of the boundary condition. (Input for <code>cg_boco_write</code> ; <i>output</i> for <code>cg_boco_info</code>)

Mid-Level Library

<code>bocotype</code>	Type of boundary condition defined. See the eligible types for <code>BCType_t</code> in Section 2.4 . Note that if <code>bocotype</code> is <code>FamilySpecified</code> the boundary condition type is being specified for the family to which the boundary belongs. The boundary condition type for the family may be read and written using <code>cg_fambc_read</code> and <code>cg_fambc_write</code> , as described in Section 16.3 . (Input for <code>cg_boco_write</code> ; output for <code>cg_boco_info</code>)
<code>ptset_type</code>	The extent of the boundary condition may be defined using a range of points or cells, or using a discrete list of all points or cells at which the boundary condition is applied. If a range of points or cells is used, <code>ptset_type</code> is set to <code>PointRange</code> . When a discrete list of points or cells is used, <code>ptset_type</code> equals <code>PointList</code> . (Input for <code>cg_boco_write</code> ; output for <code>cg_boco_info</code>)
<code>npnts</code>	Number of points in the point set defining the boundary condition region. For a <code>ptset_type</code> of <code>PointRange</code> , <code>npnts</code> is always two. For a <code>ptset_type</code> of <code>PointList</code> , <code>npnts</code> is the number of points in the <code>PointList</code> . (Input for <code>cg_boco_write</code> ; output for <code>cg_boco_info</code>)
<code>pnts</code>	Array of points defining the boundary condition region. (Input for <code>cg_boco_write</code> ; output for <code>cg_boco_read</code>)
<code>NormalIndex</code>	Index vector indicating the computational coordinate direction of the boundary condition patch normal. (Input for <code>cg_boco_normal_write</code> ; output for <code>cg_boco_info</code>)
<code>NormalListFlag</code>	For <code>cg_boco_normal_write</code> , <code>NormalListFlag</code> is a flag indicating if the normals are defined in <code>NormalList</code> ; 1 if they are defined, 0 if they're not. For <code>cg_boco_info</code> , if the normals are defined in <code>NormalList</code> , <code>NormalListFlag</code> is the number of points in the patch times <code>phys_dim</code> , the number of coordinates required to define a vector in the field. If the normals are not defined in <code>NormalList</code> , <code>NormalListFlag</code> is 0. (Input for <code>cg_boco_normal_write</code> ; output for <code>cg_boco_info</code>)
<code>NormalDataType</code>	Data type used in the definition of the normals. Admissible data types for the normals are <code>RealSingle</code> and <code>RealDouble</code> . (Input for <code>cg_boco_normal_write</code> ; output for <code>cg_boco_info</code>)
<code>NormalList</code>	List of vectors normal to the boundary condition patch pointing into the interior of the zone. (Input for <code>cg_boco_normal_write</code> ; output for <code>cg_boco_read</code>)
<code>ndataset</code>	Number of boundary condition datasets for the current boundary condition. (Output)
<code>ier</code>	Error status. (Output)

14.2 Boundary Condition Datasets

Node: BCDataSet_t

Functions	Modes
<i>ier</i> = cg_dataset_write(int fn, int B, int Z, int BC, char *DatasetName, BCType_t BCType, int *Dset);	- w m
<i>ier</i> = cg_dataset_read(int fn, int B, int Z, int BC, int DSet, char *DatasetName, BCType_t *BCType, int *DirichletFlag, int *NeumannFlag);	r - m
call cg_dataset_write_f(fn, B, Z, BC, DatasetName, BCType, Dset, <i>ier</i>)	- w m
call cg_dataset_read_f(fn, B, Z, BC, DSet, DatasetName, BCType, DirichletFlag, NeumannFlag, <i>ier</i>)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
BC	Boundary condition index number, where $1 \leq BC \leq \text{nbocos}$. (Input)
Dset	Dataset index number, where $1 \leq Dset \leq \text{ndataset}$. (Input for cg_dataset_read; output for cg_dataset_write)
DatasetName	Name of dataset. (Input for cg_dataset_write; output for cg_dataset_read)
BCType	Boundary condition type for the dataset. (Input for cg_dataset_write; output for cg_dataset_read)
DirichletFlag	Flag indicating if the dataset contains Dirichlet data. (Output)
NeumannFlag	Flag indicating if the dataset contains Neumann data. (Output)
ier	Error status. (Output)

14.3 Boundary Condition Data

Node: BCData_t

Functions	Modes
<i>ier</i> = cg_bcddata_write(int fn, int B, int Z, int BC, int Dset, BCDataType_t BCDataType);	- w m
call cg_bcddata_write_f(fn, B, Z, BC, Dset, BCDataType, <i>ier</i>)	- w m

Mid-Level Library

Input/Output

<code>fn</code>	CGNS file index number. (Input)
<code>B</code>	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
<code>Z</code>	Zone index number, where $1 \leq Z \leq \text{nzones}$. (Input)
<code>BC</code>	Boundary condition index number, where $1 \leq BC \leq \text{nbocos}$. (Input)
<code>Dset</code>	Dataset index number, where $1 \leq \text{Dset} \leq \text{ndataset}$. (Input)
<code>BCDataType</code>	Type of boundary condition in the dataset. Admissible boundary condition types are <code>Dirichlet</code> and <code>Neumann</code> . (Input)
<code>ier</code>	Error status. (Output)

15 Equation Specification

15.1 Flow Equation Set

Node: FlowEquationSet_t

Functions	Modes
<code>ier = cg_equationset_write(int EquationDimension);</code>	- w m
<code>ier = cg_equationset_read(int *EquationDimension, int *GoverningEquationsFlag, int *GasModelFlag, int *ViscosityModelFlag, int *ThermalConductModelFlag, int *TurbulenceClosureFlag, int *TurbulenceModelFlag);</code>	r - m
<code>ier = cg_equationset_chemistry_read(int *ThermalRelaxationFlag, int *ChemicalKineticsFlag);</code>	r - m
<code>call cg_equationset_write_f(EquationDimension, ier)</code>	- w m
<code>call cg_equationset_read_f(EquationDimension, GoverningEquationsFlag, GasModelFlag, ViscosityModelFlag, ThermalConductModelFlag, TurbulenceClosureFlag, TurbulenceModelFlag, ier)</code>	r - m
<code>call cg_equationset_chemistry_read_f(ThermalRelaxationFlag, ChemicalKineticsFlag, ier)</code>	r - m

Input/Output

EquationDimension	Dimensionality of the governing equations; it is the number of spatial variables describing the flow. (Input for <code>cg_equationset_write</code> ; <i>output</i> for <code>cg_equationset_info</code>)
GoverningEquationsFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the governing equations; 0 if it doesn't, 1 if it does. (<i>Output</i>)
GasModelFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the gas model; 0 if it doesn't, 1 if it does. (<i>Output</i>)
ViscosityModelFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the viscosity model; 0 if it doesn't, 1 if it does. (<i>Output</i>)
ThermalConductModelFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the thermal conductivity model; 0 if it doesn't, 1 if it does. (<i>Output</i>)
TurbulenceClosureFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the turbulence closure; 0 if it doesn't, 1 if it does. (<i>Output</i>)

Mid-Level Library

TurbulenceModelFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the turbulence model; 0 if it doesn't, 1 if it does. (<i>Output</i>)
ThermalRelaxationFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the thermal relaxation model; 0 if it doesn't, 1 if it does. (<i>Output</i>)
ChemicalKineticsFlag	Flag indicating whether or not this FlowEquationSet_t node includes the definition of the chemical kinetics model; 0 if it doesn't, 1 if it does. (<i>Output</i>)
ier	Error status. (<i>Output</i>)

15.2 Governing Equations

Node: GoverningEquations_t

Functions	Modes
<i>ier</i> = cg_governing_write(GoverningEquationsType_t Equationstype);	- w m
<i>ier</i> = cg_governing_read(GoverningEquationsType_t *EquationsType);	r - m
<i>ier</i> = cg_diffusion_write(int *diffusion_model);	- w m
<i>ier</i> = cg_diffusion_read(int *diffusion_model);	r - m
call cg_governing_write_f(EquationsType, <i>ier</i>)	- w m
call cg_governing_read_f(EquationsType, <i>ier</i>)	r - m
call cg_diffusion_write_f(diffusion_model, <i>ier</i>)	- w m
call cg_diffusion_read_f(diffusion_model, <i>ier</i>)	r - m

Input/Output

EquationsType	Type of governing equations. The admissible types are Null, UserDefined, FullPotential, Euler, NSLaminar, NSTurbulent, NSLaminarIncompressible, and NSTurbulentIncompressible. (<i>Input</i> for cg_governing_write; <i>output</i> for cg_governing_read)
diffusion_model	Flags defining which diffusion terms are included in the governing equations. This is only applicable to the Navier-Stokes equations with structured grids. See the discussion of GoverningEquations_t in the SIDS manual for details. (<i>Input</i> for cg_diffusion_write; <i>output</i> for cg_diffusion_read)
ier	Error status. (<i>Output</i>)

15.3 Auxiliary Models

Nodes: GasModel_t, ViscosityModel_t, ThermalConductivityModel_t, TurbulenceClosure_t, TurbulenceModel_t, ThermalRelaxationModel_t, ChemicalKineticsModel_t

Functions	Modes
<code>ier = cg_model_write(char *ModelLabel, ModelType_t ModelType);</code>	- w m
<code>ier = cg_model_read(char *ModelLabel, ModelType_t *ModelType);</code>	r - m
<code>call cg_model_write_f(ModelLabel, ModelType, ier)</code>	- w m
<code>call cg_model_read_f(ModelLabel, ModelType, ier)</code>	r - m

Input/Output

ModelLabel The CGNS label for the model being defined. The models supported by CGNS are:

- GasModel_t
- ViscosityModel_t
- ThermalConductivityModel_t
- TurbulenceClosure_t
- TurbulenceModel_t
- ThermalRelaxationModel_t
- ChemicalKineticsModel_t

(Input)

ModelType One of the model types (listed below) allowed for the ModelLabel selected. (Input for `cg_model_write`; *output* for `cg_model_read`)

ier Error status. (*Output*)

The types allowed for the various models are:

GasModel_t	Null, UserDefined, Ideal, VanderWaals, CaloricallyPerfect, ThermallyPerfect, ConstantDensity, RedlichKwong
ViscosityModel_t	Null, UserDefined, Constant, PowerLaw, SutherlandLaw
ThermalConductivityModel_t	Null, UserDefined, PowerLaw, SutherlandLaw, ConstantPrandtl
TurbulenceModel_t	Null, UserDefined, Algebraic_BaldwinLomax, Algebraic_CebeciSmith, HalfEquation_JohnsonKing, OneEquation_BaldwinBarth, OneEquation_SpalartAllmaras, TwoEquation_JonesLaunder, TwoEquation_MenterSST, TwoEquation_Wilcox
TurbulenceClosure_t	Null, UserDefined, EddyViscosity, ReynoldsStress, ReynoldsStressAlgebraic
ThermalRelaxationModel_t	Null, UserDefined, Frozen, ThermalEquilib, ThermalNonequilib
ChemicalKineticsModel_t	Null, UserDefined, Frozen, ChemicalEquilibCurveFit, ChemicalEquilibMinimization, ChemicalNonequilib

16 Families

16.1 Family Definition

Node: Family_t

Functions	Modes
<i>ier</i> = cg_family_write(int fn, int B, char *FamilyName, int *Fam);	- w m
<i>ier</i> = cg_nfamilies(int fn, int B, int *nfamilies);	r - m
<i>ier</i> = cg_family_read(int fn, int B, int Fam, char *FamilyName, int *nFamBC, int *nGeo);	r - m
call cg_family_write_f(fn, B, FamilyName, Fam, ier)	- w m
call cg_nfamilies_f(fn, B, nfamilies, ier)	r - m
call cg_family_read_f(fn, B, Fam, FamilyName, nFamBC, nGeo, ier)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
nfamilies	Number of families in base B. (Output)
Fam	Family index number, where $1 \leq \text{Fam} \leq \text{nfamilies}$. (Input for cg_family_read; output for cg_family_write)
FamilyName	Name of the family. (Input for cg_family_write; output for cg_family_read)
nFamBC	Number of boundary conditions for this family. This should be either 0 or 1. (Output)
nGeo	Number of geometry references for this family. (Output)
ier	Error status. (Output)

16.2 Geometry Reference

Node: GeometryReference_t

Functions	Modes
<code>ier = cg_geo_write(int fn, int B, int Fam, char *GeoName, char *FileName, char *CADSystem, int *G);</code>	- w m
<code>ier = cg_geo_read(int fn, int B, int Fam, int G, char *GeoName, char **FileName, char *CADSystem, int *nparts);</code>	r - m
<code>ier = cg_part_write(int fn, int B, int Fam, int G, char *PartName, int *P);</code>	- w m
<code>ier = cg_part_read(int fn, int B, int Fam, int G, int P, char *PartName);</code>	r - m
<code>call cg_geo_write_f(fn, B, Fam, GeoName, FileName, CADSystem, G, ier)</code>	- w m
<code>call cg_geo_read_f(fn, B, Fam, G, GeoName, FileName, CADSystem, nparts, ier)</code>	r - m
<code>call cg_part_write_f(fn, B, Fam, G, PartName, P, ier)</code>	- w m
<code>call cg_part_read_f(fn, B, Fam, G, P, PartName, ier)</code>	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Fam	Family index number, where $1 \leq \text{Fam} \leq \text{nfamilies}$. (Input)
G	Geometry reference index number, where $1 \leq G \leq \text{nGeo}$. (Input for <code>cg_geo_read</code> , <code>cg_part_write</code> , <code>cg_part_read</code> ; <i>output</i> for <code>cg_geo_write</code>)
P	Geometry entity index number, where $1 \leq P \leq \text{nparts}$. (Input for <code>cg_part_read</code> ; <i>output</i> for <code>cg_part_write</code>)
GeoName	Name of GeometryReference_t node. (Input for <code>cg_geo_write</code> ; <i>output</i> for <code>cg_geo_read</code>)
FileName	Name of geometry file. (Input for <code>cg_geo_write</code> ; <i>output</i> for <code>cg_geo_read</code>)
CADSystem	Geometry format. (Input for <code>cg_geo_write</code> ; <i>output</i> for <code>cg_geo_read</code>)
nparts	Number of geometry entities. (<i>Output</i>)
PartName	Name of a geometry entity in the file FileName. (Input for <code>cg_part_write</code> ; <i>output</i> for <code>cg_part_read</code>)
ier	Error status. (<i>Output</i>)

16.3 Family Boundary Condition

Node: FamilyBC_t

Functions	Modes
<i>ier</i> = cg_fambc_write(int fn, int B, int Fam, char *FamBCName, BCType_t BCType, int *BC);	- w m
<i>ier</i> = cg_fambc_read(int fn, int B, int Fam, int BC, char *FamBCName, BCType_t *BCType);	r - m
call cg_fambc_write_f(fn, B, Fam, FamBCName, BCType, BC, <i>ier</i>)	- w m
call cg_fambc_read_f(fn, B, Fam, BC, FamBCName, BCType, <i>ier</i>)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Fam	Family index number, where $1 \leq \text{Fam} \leq \text{nfamilies}$. (Input)
BC	Family boundary condition index number. This must be equal to 1. (Input for cg_fambc_read; <i>output</i> for cg_fambc_write)
FamBCName	Name of the FamilyBC_t node. (Input for cg_fambc_write; <i>output</i> for cg_fambc_read)
BCType	Boundary condition type for the family. See the eligible types for BCType_t in Section 2.4. (Input for cg_fambc_write; <i>output</i> for cg_fambc_read)
ier	Error status. (<i>Output</i>)

16.4 Family Name

Node: FamilyName_t

Functions	Modes
<i>ier</i> = cg_famname_write(char *FamilyName);	- w m
<i>ier</i> = cg_famname_read(char *FamilyName);	r - m
call cg_famname_write_f(FamilyName, <i>ier</i>)	- w m
call cg_famname_read_f(FamilyName, <i>ier</i>)	r - m

Input/Output

FamilyName	Family name. (Input for cg_famname_write; <i>output</i> for cg_famname_read)
ier	Error status. (<i>Output</i>)

17 Time-Dependent Data

17.1 Base Iterative Data

Node: BaseIterativeData_t

Functions	Modes
<i>ier</i> = cg_biter_write(int fn, int B, char *BaseIterName, int Nsteps);	- w m
<i>ier</i> = cg_biter_read(int fn, int B, char *BaseIterName, int *Nsteps);	r - m
call cg_biter_write_f(fn, B, BaseIterName, Nsteps, <i>ier</i>)	- w m
call cg_biter_read_f(fn, B, BaseIterName, Nsteps, <i>ier</i>)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
BaseIterName	Name of the BaseIterativeData_t node. (Input for cg_biter_write; <i>output</i> for cg_biter_read)
Nsteps	Number of time steps or iterations. (Input for cg_biter_write; <i>output</i> for cg_biter_read)
ier	Error status. (<i>Output</i>)

17.2 Zone Iterative Data

Node: ZoneIterativeData_t

Functions	Modes
<i>ier</i> = cg_ziter_write(int fn, int B, int Z, char *ZoneIterName);	- w m
<i>ier</i> = cg_ziter_read(int fn, int B, int Z, char *ZoneIterName);	r - m
call cg_ziter_write_f(fn, B, Z, ZoneIterName, <i>ier</i>)	- w m
call cg_ziter_read_f(fn, B, Z, ZoneIterName, <i>ier</i>)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Family index number, where $1 \leq Z \leq \text{nzones}$. (Input)
ZoneIterName	Name of the ZoneIterativeData_t node. (Input for cg_ziter_write; <i>output</i> for cg_ziter_read)
ier	Error status. (<i>Output</i>)

17.3 Rigid Grid Motion

Node: RigidGridMotion_t

Functions	Modes
<i>ier</i> = cg_rigid_motion_write(int fn, int B, int Z, char *RigidGridMotionName, RigidGridMotionType_t RigidGridMotionType, int *R);	- w m
<i>ier</i> = cg_n_rigid_motions(int fn, int B, int Z, int *n_rigid_motions);	r - m
<i>ier</i> = cg_rigid_motion_read(int fn, int B, int Z, int R, char *RigidGridMotionName, RigidGridMotionType_t RigidGridMotionType);	r - m
call cg_rigid_motion_write_f(fn, B, Z, RigidGridMotionName, RigidGridMotionType, R, <i>ier</i>)	- w m
call cg_n_rigid_motions_f(fn, B, Z, n_rigid_motions, <i>ier</i>)	r - m
call cg_rigid_motion_read_f(fn, B, Z, R, RigidGridMotionName, RigidGridMotionType, <i>ier</i>)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Family index number, where $1 \leq Z \leq \text{nzones}$. (Input)
RigidGridMotionName	Name of the RigidGridMotion_t node. (Input for cg_rigid_motion_write; output for cg_rigid_motion_read)
RigidGridMotionType	Type of rigid grid motion. The admissible types are Null, UserDefined, ConstantRate, and VariableRate. (Input for cg_rigid_motion_write; output for cg_rigid_motion_read)
n_rigid_motions	Number of RigidGridMotion_t nodes under zone Z. (Output)
R	Rigid rotation index number, where $1 \leq R \leq \text{n_rigid_motions}$. (Input for cg_rigid_motion_read; output for cg_rigid_motion_write)
ier	Error status. (Output)

17.4 Arbitrary Grid Motion

Node: ArbitraryGridMotion_t

Functions	Modes
<i>ier</i> = cg_arbitrary_motion_write(int fn, int B, int Z, char *ArbitraryGridMotionName, ArbitraryGridMotionType_t ArbitraryGridMotionType, int *A);	- w m
<i>ier</i> = cg_n_arbitrary_motions(int fn, int B, int Z, int *n_arbitrary_motions);	r - m
<i>ier</i> = cg_arbitrary_motion_read(int fn, int B, int Z, int A, char *ArbitraryGridMotionName, ArbitraryGridMotionType_t ArbitraryGridMotionType);	r - m
call cg_arbitrary_motion_write_f(fn, B, Z, ArbitraryGridMotionName, ArbitraryGridMotionType, A, <i>ier</i>)	- w m
call cg_n_arbitrary_motions_f(fn, B, Z, n_arbitrary_motions, <i>ier</i>)	r - m
call cg_arbitrary_motion_read_f(fn, B, Z, A, ArbitraryGridMotionName, ArbitraryGridMotionType, <i>ier</i>)	r - m

Input/Output

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq B \leq \text{nbases}$. (Input)
Z	Family index number, where $1 \leq Z \leq \text{nzones}$. (Input)
ArbitraryGridMotionName	Name of the ArbitraryGridMotion_t node. (Input for cg_arbitrary_motion_write; <i>output</i> for cg_arbitrary_motion_read)
ArbitraryGridMotionType	Type of arbitrary grid motion. The admissible types are Null, UserDefined, NonDeformingGrid, and DeformingGrid. (Input for cg_arbitrary_motion_write; <i>output</i> for cg_arbitrary_motion_read)
n_arbitrary_motions	Number of ArbitraryGridMotion_t nodes under zone Z. (<i>Output</i>)
A	Arbitrary grid motion index number, where $1 \leq A \leq \text{n_arbitrary_motions}$. (Input for cg_arbitrary_motion_read; <i>output</i> for cg_arbitrary_motion_write)
ier	Error status. (<i>Output</i>)

18 Links

Functions	Modes
<code>ier = cg_link_write(char *nodename, char *filename, char *name_in_file);</code>	- w m
<code>ier = cg_is_link(int *path_length);</code>	r - m
<code>ier = cg_link_read(char **filename, char **link_path);</code>	r - m
call <code>cg_link_write_f(nodename, filename, name_in_file, ier)</code>	- w m
call <code>cg_is_link_f(path_length, ier)</code>	r - m
call <code>cg_link_read_f(filename, link_path, ier)</code>	r - m

Input/Output

<code>nodename</code>	Name of the link node to create, e.g., GridCoordinates. (<i>Input</i>)
<code>filename</code>	Name of the linked file, or empty string if the link is within the same file. (<i>Input</i> for <code>cg_link_write</code> ; <i>output</i> for <code>cg_link_read</code>)
<code>name_in_file</code>	Path name of the node which the link points to. This can be a simple or a compound name, e.g., Base/Zone 1/GridCoordinates. (<i>Input</i>)
<code>path_length</code>	Length of the path name of the linked node. The value 0 is returned if the node is not a link. (<i>Output</i>)
<code>link_path</code>	Path name of the node which the link points to. (<i>Output</i>)
<code>ier</code>	Error status. (<i>Output</i>)

Use `cg_goto(_f)`, described in [Section 4](#), to position to a location in the file prior to calling these routines.

When using `cg_link_write`, the node being linked to does not have to exist when the link is created. However, when the link is used, an error will occur if the linked-to node does not exist.

Only nodes that support child nodes will support links.

Memory is allocated by the library for the return values of the C function `cg_link_read`. This memory should be freed by the user when no longer needed.